CLAS-NOTE 97-003
Date : 1997/05/1314 : 33 : 39
A. Freyberger
M. Ito
CVS file: docs/cms.tex
Revision : 1.19

CLAS Offline Code Management

Abstract

A new code management scheme based on CVS, GNU Make and a simple directory structure is documented. Offsite users will be able to check out code directly from the JLab-CLAS code repository and check the modified code back in seamlessly. A makefile scheme (using GNU Make) has also been implemented that will support the casual user, code developer, and multiple platforms.

1 Introduction

A complete code management system is made up of three components, a directory structure, a make system and a revision control system. For the new CLAS code management system we have developed a simple directory structure and a make system that incorporates many of the ideas outlined in the GNU Make manual [1]. We have chosen to use CVS as the revision control system since it will support remote code developing and has a wide user base.

Since the scope of CLAS code will only grow in the next few years, the make system and directory structure must accommodate growth. This not only includes new code but also new platforms. Dynamic systems help by minimizing the number of changes needed to accommodate growth. This philosophy has driven the design. To add a new software package, only the directory and the code need to be added to the repository. The Makefile will dynamically locate the source in the directory, C and/or FORTRAN, build the dependencies, compile the code and archive the object files into a library. Include or header files in the source directory, in a directory parallel to the source directory or in a directory immediately beneath the source directory are located, added to the dependency list and those directories are fed to the CPP preprocessor automatically. One side effect of this dynamism is that the make system is directory driven, names of the created library and binary are derived from the directory from which `make` is executed. Internal include dependencies are generated automatically via a combination of `gcc` and `sed` and are updated whenever the source or its dependencies have been modified.

Just as names for binaries and libraries are derived from the directory names, the platform dependence is keyed off the response of the UNIX `uname` command. For example the `uname` for a Sun operating system is `SunOS` and binaries are ported to a directory *bin/SunOS* and libraries are found in *lib/SunOS*. These platform specific directories are created automatically by the make system. On the JLab CUE system this means that a make command executed in the same directory but on different machines will will produce binaries or libraries in different directories. To add a new platform only two files need to be created (templates based on other platforms are available). Once these machine specific parts of the make system are worked out for a system, they are checked into the repository making them available to other users of that platform.

Often a major drawback of complicated code management systems is that the system itself is unmanageable and user unfriendly. These problems can be avoided by a lean system, that is user customizable and allows for exceptions. For example developers are strongly encouraged to use the make system developed, however custom makefiles are allowed. Documentation also plays a strong role in the success of a code management system and we hope this document gets things off on the right note.

A big change over the previous system is that we expect collaborating institutions to build the entire CLAS software on their local computers, whereas in the previous system compiled libraries and binaries were made available. Only the source and the makefiles will be distributed. The advantage of this method is that you are guaranteed to get binaries that work on your system and the transfers are faster. Once set up properly, weekly offsite updates and builds can be easily automated. Since it would be a wasted effort to have several builds at the same offsite computer it would be best if each institution designate one person to build the entire system locally. Once this build is complete the individual members can link against it. The advantage of this method is that the JLab librarian deals with one offsite librarian per institution.

The system has tried to utilize the strengths of CVS and GNU Make and we have created a system that does not use any shell scripts or other tools. This will make it easier to maintain. We hope this moves CLAS software forward and we really need feedback from offsite users to know how well it is working for offsite builds.

The next section lists the installed software requirements for using the system. The section after that comprises a set of quick tutorials for the various types of users. After the tutorials, there are various **gory details** sections which will describe the directory tree structure, makefile system and code management system. Hopefully you will only need to refer to the **gory details** sections once you have moved beyond the new user phase. All collaborators should obtain a copy of the GNU Make manual [1] and the CVS manual [2], both of which are free and available on the net from the Free Software Foundation. There are also pointers to these documents on the CLAS Offline Software Web Page, `http://www.cebaf.gov/~manak/offline.html`.

A note on the format used in this document. All UNIX environment variables will be UPPER CASE CHARACTERS encased in () or preceded by a $, for example $CLAS_LIB denotes an environment variable that points to the compiled CLAS libraries. Text that is to be entered at the keyboard or listings of code will be denoted by the typewriter font, for example: `make lib` or `ls $CLAS_LIB`. Files and directories will be *italicized* as in, edit your *.login* file or more typically, edit *.login* then `source .login`.

2 Minimal Requirements to Build

Before you start to build the entire system you should make sure you have the following software on your machine;

- `gnu-make`
- `cvs` version 1.9 or higher
- `gcc` (needed for dependency creation)
- `tcl/tk` (needed by `recsis`)
- `cernlib` (needed by everybody)
- Motif libraries (needed by `ced` and `gsim_int`)
- `FORTRAN` compiler
- `C` compiler

CLAS software has been tested on the following platforms and Makefiles exist for these platforms:

- Linux (robust)
- HP-UX (robust)
- SunOS (robust)
- AIX (robust)
- OSF1 (in development)

If you are not working on one of these systems you will need to look at the Makefile gory detail section before proceeding.

3 Quick Start Tutorials

We expect that there will be three "hats" that you might wear when using the system.

User You want to analyze data or Monte Carlo using personal analysis code. You want to make histograms, play with cuts, measure efficiencies, etc. You are not interested in changing the routines that reside on the tree.

Librarian You want to create a complete build of the system for others to use or perhaps even for personal use. Again you do not want to change the pre-existing routines. *

Developer You want to fix or improve one of the code packages on the tree or want to add a new package.

The following sections give examples of tasks for each hat style.

One quick note on CVS. Librarians and Developers will have to use CVS to accomplish their tasks (Users don't have to). If the computer you use does not have direct access to the file server at JLab, there will we be an extra step needed before executing CVS commands to allow CVS to access the repository. The step is

```
cvs login
password: ********
```

where password is that which corresponds to the username found in the $CVSROOT environment variable (in the JLab CUE environment). More on $CVSROOT below.

3.1 User Tutorial

We have to assume that for you to function as a User, some Librarian has already made a complete build of the system on a local disk, and has customized a short script that should be included in your .cshrc (we will assume

*Each institution should assign a librarian to do an initial build and subsequent updates to keep the libraries up to date. If such a librarian is identified we will make every effort to notify her or him about modifications.

C shell use throughout). For definiteness, let us say you are at Pitt, working on a HP-UX system and the build has been done under the directory /home/clas/build. Let us say that you want to make a RECSIS executable that calls three of your routines, *xyz_init.F, xyz_event.F* and *xyz_last.F*, called at job begin, on every event, and at job end respectively. Further, *xyz_event.F* and *xyz_last.F* use an include file *xyz.inc*. Finally, let's assume that all four of these files sit in the directory */home/username/xyz*

1. Edit your .cshrc file to include the line †

   ```
   source /home/clas/build/packages/cms/pitt.cshrc
   ```

   This is the Librarian-provided script mentioned above. Now

   ```
   source .cshrc
   ```

2. Get the RECSIS user code.

   ```
   cd /home/username/xyz
   cp $CLAS_PACK/user/* .
   ```

   You will get a warning when the command tries to copy a directory file, *CVS*. That is ok. You don't want that directory. Now your directory looks like:

   ```
   > ls
   Makefile          user_evnt.F       user_xtra.F
   user_bevt.F       user_init.F       xyz.inc
   user_brun.F       user_last.F       xyz_event.F
   user_control.inc  user_sda.F        xyz_finish.F
   user_erun.F       user_tcl_init.F   xyz_init.F
   ```

3. Edit the user routines to call your private routines. *user_init* should call *xyz_init*, *user_event* should call *xyz_event*, and *user_last* should call *xyz_last*.

† The old system used `source /apps/clas/u1/etc/profile/.clascshrc`. Comment this out of your .cshrc to use the new system.

4. Make the object library and executable. Type

   ```
   make lib exe
   ```

   or simply

   ```
   make
   ```

   As this implies, the default targets are `lib` and `exe`. An object library and an executable will be made, */home/username/lib/HP-UX/libxyz.a* and */home/username/medbreak/bin/HP-UX/xyz* respectively. The latter is a version of RECSIS incorporating your private *user* and *xyz* routines. The name *xyz* came from the name of directory where `make` was invoked.

5. To run the executable do the following:

   ```
   /home/username/bin/HP-UX/xyz -t $CLAS_PACK/tcl/newinit
   ```

   or if */home/username/bin/HP-UX* is already in your path and you have a local version of *newinit.tcl*

   ```
   xyz -t newinit
   ```

   newinit.tcl is the standard tcl script used to set up RECSIS. You should copy and edit *newinit.tcl* to contain the appropriate input and output filenames as well as initialization parameters, and invoke your local version. The RECSIS manual explains the syntax further.

3.2 Librarian Tutorial

Again let us use the Pitt example. Assume we want to do a build under the directory */home/clas/build* starting from scratch.

1. Set the $CVSROOT environment variable.

   ```
   setenv CVSROOT \
    :pserver:username@jlabs1.cebaf.gov:/apps/clas/u1/clas_cvs
   ```

`username` must have "clas" as its default group id. At this writing, you must use jlabs1 as the CVS server. `/apps/clas/u1/clas_cvs` is sufficient at JLab.

2. Do the CVS login.

   ```
   cvs login
   ```

   Recall that this can be omitted at JLab.

3. Checkout the tree from the repository at JLab.

   ```
   cd /home/clas/build    # build here
   cvs checkout packages  # get the tree over the internet
   ```

4. Compile and link everything.

   ```
   cd packages  # go into the first level of the tree
   make         # build it
   ```

   A complete tree as described in the following chapters will be created. Executables will be in */usr/home/clas/build/bin/HP-UX* and all of the object libraries will be in */usr/home/clas/build/build/HP-UX*. Note that following this procedure does not put the libraries and executables in the same place as in the User tutorial. A different makefile was used there.

5. Create a *pitt.cshrc*.

   ```
   cd /home/clas/build/packages/cms
   cp offsite.cshrc pitt.cshrc
   ```

   Now edit *pitt.cshrc* to reflect local conditions. The file is self-documenting. Since $CVSROOT is defined in this file, if pitt.cshrc already exists and has been `source`d, step 1 can be omitted.

6. Check the new file in to CVS repository.

```
cvs add pitt.cshrc # add new files before commit
cvs commit -m "CLAS .cshrc file for Panthers" pitt.cshrc
```

The `-m` switch is for a mandatory comment.

The last two steps need only be done once. Subsequent checkouts will pull out *pitt.cshrc* along with everything else.

3.3 Developer Tutorial

Again assume that you start with a complete build in */home/clas/build* and you want to work on the dc code.

1. Get the CLAS environment:

   ```
   source /home/clas/build/packages/cms/pitt.cshrc
   ```

   in your *.cshrc*.

2. Make a work directory.

   ```
   mkdir /home/username/work
   cd /home/username/work
   ```

3. Log into the repository. (Omit this step at JLab.)

   ```
   cvs login
   password: ********
   ```

4. Checkout the dc package.

   ```
   cvs checkout dc
   ```

   At this point, if you `cd dc`, you will see (as of this writing)

```
> ls
CVS                       dc_read_xvst.F
Makefile                  dc_tcl_init.F
dc_brun.F                 dc_time_to_distance.F
dc_distance_to_time.F     dc_xvst.inc
dc_fill_d_to_t_table.F    dc_xvst_curve.F
dc_fill_t_to_d_table.F    dc_xvst_get_doca.F
dc_getcal_geom.F          dc_xvst_get_tdrift.F
dc_getcal_tdly.F          only_rot.F
dc_getcal_xvst.F          trans_rot.F
dc_read_geom.F            w_pos_dir.F
dc_read_tdly.F
```

5. Make your changes to the code.

6. Make a library using the modified code.

   ```
   make lib
   ```

   /usr/username/lib/HP-UX/libdc.a is created.

7. Make an executable using the new library.

   ```
   make exe
   ```

   /usr/username/bin/HP-UX/dc is created. Again, as in the case with User, the name of the executable came from the directory name where `make` was invoked.

8. Test your changes!

9. Check in your changes. In the dc directory type, for example

   ```
   cvs commit -m "Really improved the code."
   ```

   All files that have changed and are under CVS control will have a new revision added in the repository at JLab. This step could have been done file-by-file by repeating the commit with an explicit file name at the end of the command, one command per file. That way you get file-specific comments.

10. Release your copy of the package.

```
> cd .. # You are now in /home/username/work
> cvs release -d dc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module 'dc': y
```

The directory *dc* and all of its files have been removed. If you had forgotten to commit a file, the release command would have alerted you and you could have aborted the release (and delete).

In this example we checked out only one package. Often a change in one package forces modification of another. In that case you check out all of the packages that need modification, make all of the new libraries and make the executable from the directory of your choice, again depending on what you want the name of the executable to be.

4 Directory Structure

In this section we describe a complete build of the system. Usually there will be more than one such build on a given system, reflecting different states of the code. This variety of builds will be discussed in the section on code management. Let us assume that our build was done below the subdirectory *build*:

```
build/
     |--bin/
     |     |--SunOS/  # Sun Binaries
     |     |       |--recsis
     |     |       |--gsim_bat
     |     |       |--gsim_int
     |     |
     |     |--HP-UX/  # HP-UX binaries
     |     |--AIX/    # AIX binaries
     |
     |--lib/
     |     |--SunOS/  # Sun libraries
```

```
|     |       |--libdc.a
|     |       |--libec.a
|     |       |--librecsis.a
|     |       |--recsis.o
|     |       |--...         # more libraries and objects
|     |
|     |--HP-UX/  # HP-UX libraries
|     |--AIX/    # AIX libraries
|
|--packages/      # Top of the source tree
```

Underneath `build` there are three subdirectories, bin, lib and packages. bin contains executables, lib contains object libraries and individual object modules, and packages contains the source and the makefiles. Note that in this example, there are three platforms supported, as reflected by the directories SunOS, HP-UX and AIX underneath bin and lib.

Going down into packages we see:

```
packages--|
          |--cms      # Makefiles found here
          |--bankdefs # ddl files
          |--tcl      # tcl/tk scripts
          |--scripts  # useful scripts
          |--include  # multi-package include files
          |--dc       # source for libdc.a
          |--sc       # source for libcc.a
          |--bos      # source for libbos.a
          |--...      # more packages
          |--user     # source for libuser.a
```

Each of these directories contains all of the source (**.F* and **.c* files) and include files (**.inc* and **.h* files) used in a particular package. dc for the drift chamber, sc for the time-of-flight, etc. Each package corresponds to one of the object libraries in the lib directory, e. g., the dc package creates *libdc.a*. Exceptions to this are described below. Also each directory (including the exceptional directories) contains a makefile called Makefile. Finally each non-exceptional directory contains a directory called depends that contains dependency files used by the make system. These files insure that make will recompile an unchanged source code file if one of the files it includes changes.

A few of the directories contain main programs and so have extra references to them made in the makefile system. At present these are recsis, gsim_bat and gsim_int.

The exceptions to the "analysis packages" description are:

cms: Contains makefiles and include files for makefiles.

bankdefs: Contains the bank definition files. These are not source code files, rather they act more as data.

include: This directory contains only those include files that are shared among more than one package. If an include file is used internally by a single package, it should go in the directory for that package.

All of the exceptional directories contain a dummy makefile which does nothing.

As an example of an analysis packages, here is the *ec* directory:

```
ec/
  |--Makefile
  |--EcCal.CMN
  |--EcDba.CMN
  |--...              # more include files
  |--Ec\_general.PAR
  |--ec\_brun.F
  |--ec\_dalitz.F
  |--ec\_evnt.F
  |--...              # more FORTRAN source files
  |--depends/
            |--AIX/              # dependencies for AIX
            |     |--ec\_brun.d
            |     |--ec\_dalitz.d
            |     |--ec\_evnt.d
            |     |--...         # more dependency files
            |
            |--HP-UX/
```

```
|       |--...       # dependencies for HP
|
|--SunOs/
        |--...       # dependencies for Sun
```

4.1 Additional Directories

The tree as built does not contain the *parms* directory, the calibration files or a documentation directory. These must be installed separately by the Librarian. Those procedures are properly beyond the scope of this document.

5 The Makefile System

Make is a standard UNIX utility to generate the commands needed to compile source, group the resulting object modules into object libraries and link together executables from those libraries. Each application of the make program is different in detail. This section describes the system that we use.

5.1 Philosophy

Recall that part of the philosophy of the system is to avoid explicit lists of source files in the makefiles. Rather the makefiles look in the appropriate directories and make these lists on the fly. This makes maintenance easier; there is one less step when adding or removing a source file. Also C and FORTRAN files and their include files are all kept in one directory for a given package (with the exceptions noted in the previous section). Finally, the make system must support multiple platforms. This is reflected in the directory structure, which in turn is generated from the make system, and in two other features. One is the use of C preprocessor variables in the source code (including the FORTRAN source) to flag sections of code which should be included or excluded on a given platform. See any standard textbook on C for a description of the preprocessor. The other feature is platform dependent inclusions to the makefiles. These are kept in the *packages/cms* directory and have names like `link.SunOS.mk` and `flags.Linux.mk`.

5.2 Makefiles

There are three classes of files named *Makefile*.

5.2.1 Mother of All Makefiles

There is only one file in the first class and it referred to as the "Mother of All Makefiles." It contains all the necessary macro definitions, suffix rules, target:prerequisite combinations and commands necessary to make **all** of the object modules, object libraries and executables in the system. Most of the subsequent details apply to this makefile.

5.2.2 Individual Package Makefiles

The second class has already been mentioned. These are the makefiles found in the source directory for each package. In general these only set a few packages dependent variables (and in many cases, there are not any of these) and invoke the first class.

Below is the listing of the *Makefile* found in the *user* directory. This makefile invokes *$CLAS_CMS/Makefile* using the default link list defined in *$CLAS_CMS/link.mk* and *$CLAS_CMS/link.$OS_NAME.mk* It is provided as an example:

```
# Individual package makefile, includes $CLAS_CMS/Makefile with the
# appropriate arguments

# TOP_DIR points to the final location of binary and library tree:
# $(TOP_DIR)/bin/$(OS_NAME)/       $(TOP_DIR)/lib/$(OS_NAME)/

# USERS should set the environment TOP_DIR to a directory of their
# choice to override the default relative path
# (default = /home/$USERNAME).

# CLAS_CMS points to the location of the included makefiles.
# Default is to get this from the environment.

ifndef CLAS_CMS
 CLAS_CMS = ../
endif
```

```
# define the link list

REQUIRED_OBJS=$(CLAS_LIB)/recsis.o $(CLAS_LIB)/rec_work.o

LIBNAMES  = $(RECSIS_LIBS1) $(MAIN_DIRECTORY)$(ADD_DEBUG)
LIBNAMES += $(RECSIS_LIBS2) $(RECSIS_MD_LIBS)

SHARED_LIBS=$(RECSIS_SHARED_LIBS)

# get the generic Makefile and predefined link lists;
# link.$(OS_NAME).mk contains the machine dependent part of the
# link list.

include $(CLAS_CMS)/Makefile
include $(CLAS_CMS)/link.mk
include $(CLAS_CMS)/link.$(OS_NAME).mk
```

$(CLAS_CMS)/Makefile has two targets, `lib` and `exe`. By invoking `make` the lib target will be executed followed by the exe target, resulting in a new library and a new executable.

5.2.3 Complete Build Makefile

The third class is also a single makefile. It sits in the *packages* directory (on the same level as *dc, ec, sc, bos,* etc.) This makefile is used to build all packages present in the tree. In keeping with the philosophy, it makes an inventory of the directories under packages and tries to make each of them by invoking its makefile. This is the reason for the dummy makefiles for the exceptional directories. This makefile will generally only be invoked when trying to do a build of the complete tree, although it can be used when only a subset of packages have been put in a build.

The flow chart is as follows:

- invoke `make`
- invoke *cms/clas_lib.mk*
  - make a directory listing of all the subdirectories

  - cd into each subdirectory and invoke `make lib`
    * make a listing of .c and .F files
    * compile them with the flags found in *cms/flags.$OS_NAME.mk*
    * object files are archived in *$TOP_DIR/lib/$OS_NAME/libdirectory.a*

- invoke *cms/clas_bin.mk*
  - make all the targets listed. The files *cms/link.mk* and *link.$OS_NAME.mk* contain the link list for the binaries.
  - binaries are ported to the directory, *$TOP_DIR/bin/$OS_NAME*.

Thus the library building is done by making a directory listing, the system is dynamic and new libraries are automatically added.

So compiling and linking can occur in two contexts. There is the operation that builds all libraries and all executables. This operation uses the third class of makefile and is completely automated. Then there is the case where the user builds one or several of the libraries and depends on a build of the first type to provide the other packages. The following sections will assume the second context, where documentation is really needed.

5.3 Compilation

C and FORTRAN files are compiled and put directly into object libraries. No *.o files are left over on disk after compilation is done. Include files are searched for in the following directories in the order shown:

../include the local version of the include package

./ the local directory

***/** all directories below the local directory

$CLAS_PACK/include the include package in the complete build

$CLAS_PACK/gsim/include the gsim include directory in the complete build

other non-CLAS directories

Note that you will not see -I switches in the compile command for any directory that does not exist or that does not contain a *.h or *.inc file.

5.4 Object Libraries

For the non-exceptional directories, libraries (i. e., UNIX archive files) will contain a *.o* file (only visible via a `ar -t` command) for each **.c* and **.F* file found in that directory. The name of the library is *lib<package>.a* where *<package>* is the name of the relevant directory, e. g., *libdc.a*. The library will appear in the directory *$TOP_DIR/lib/$OS_NAME*. To make a library, you type

```
cd <package>
make lib
```

If at any time you wish to use the debugger, simply add `DEBUG=yes` to the `make` command and debug versions of the libraries will be linked to make a debug version of the binary. For example:

```
cd user
make DEBUG=yes
```

will make a binary called *user_debug* in the *bin* area which has been compiled and linked with debug options (`-g`). You then can invoke `dbx, xdb, gdb` or which ever debugger is available on your system.

5.5 Executables

An executable can be made from any non-exceptional package directory. The main program defaults to *recsis*, except for executables made from the *gsim** directories. A make done from the directory *gsim_bat* uses *gsim_bat* as main, a make done from *gsim_int* uses *gsim_int*, and a make done from *gsim* makes both. The name of the executable matches the package name, e. g., *dc*, *ec*, *user*. The executable appears in *$TOP_DIR/bin/$OS_NAME*. Platform dependent modification to the link lists (i. e., the library lists) are kept in *$CLAS_CMS/link.$OS_NAME.mk*, Libraries are searched for first in the

$TOP_DIR/$OS_NAME/lib and then in *$CLAS_LIB* In this way libraries that have been built locally by the user are always used, rather than the pre-built libraries. To make an executable type:

```
cd <package>
make exe
```

The `DEBUG=yes` trick works here too.

Since the name of the executable is derived from the directory name, if you wish to call the executable something more appropriate, simply rename your (usually *user*) directory to a name of your choosing. For example, `mv user pipipi0; cd pipipi0; make` will change the your executable and user library to *pipipi0* and *libpipipi0.a* respectively.

5.6 Dependencies

Dependencies are generated automatically at compile time. A variation on the method recommended in the GNU Make manual is used. As mentioned before, this method uses the C compiler and the UNIX sed utility. It will generate some message when the **.d* files do not exist and must be created for the first time like

```
depends/HP-UX/user_evnt.d: No such file or directory
```

which users can safely ignore. The important feature of the dependency scheme is that it insures that routines are re-compiled when their include files change, even when the routine itself has not. And this is done recursively, down through the chain of included include files.

5.7 Simple Extensions

If you want to change the final location for your libraries and binaries you can set an environment variable called $TOP_DIR. For example `setenv TOP_DIR /scratch/$USERNAME` will result in your binaries and libraries being shipped to the */scratch* disk.

If you want to change the link list to include another library you will need to edit the *Makefile* and change the lines:

```
LIBNAMES=$(RECSIS_LIBS1) $(MAIN_DIRECTORY)$(ADD_DEBUG) \
  $(RECSIS_LIBS2) $(RECSIS_MD_LIBS)
```

to something like:

```
LIBNAMES=$(RECSIS_LIBS1) $(MAIN_DIRECTORY)$(ADD_DEBUG) \
  myutilities jane_utilities \
  $(RECSIS_LIBS2) $(RECSIS_MD_LIBS)
```

where *libmyutilities.a* and *libjane_utilities.a* are the new libraries. Note that you need only include the name of library, the `-l` will be added automatically by the *Makefile*.

5.8 Using the Mother of All Makefiles

As mentioned in previous most of the work in building libraries and binaries is done by a single Makefile, *$CLAS_CMS/Makefile*. This Makefile performs many functions and on the face of it looks rather complicated. At last count it is 413 lines long **but 194** of those lines are comments! This Makefile is firmly based on the GNU Make [1] book and anyone with a copy of this book should be able to figure out what each line is doing and why it is needed.

To demonstrate how to use this *cms/Makefile*, we'll use a simple example of a small package consisting of files *hexit.F hinit.F prime.F async.c ctime.c*. The **main** routine is *prime.F* which calls *hinit.F prime.F async.c ctime.c* and also uses cernlib, tcl, and clasutil libraries. In order to make a binary we need to communicate to *Makefile* two things, the main routine and the link list. This can be done at the command line or by editing *Makefile*. To define the main routine use the `CREATED_F` variable:

- `CREATED_F = prime.F`

Defining the link list is just as easy use the `LIBNAMES` variable:

```
LIBNAMES = $(MAIN_DIRECTORY)
LIBNAMES += mathlib kernlib packlib clasutil
LIBNAMES += tcl
```

Note the `LIBNAMES` variable starts with the `$MAIN_DIRECTORY` variable which is the library created from the source in the working directory. With these two modifications, the library containing *hexit.o hinit.o prime.o async.o ctime.o* is generated with a `make lib` command and an executable is generated with a `make exe` command. A complete build of library and executable is done with a simple `make` command.

Makefile includes $CLAS_CMS/*flags.$OS_NAME.mk*. This file contains the compile option flags. That means that in this example you will need to define $CLAS_CMS to point the default copy of *flags.$OS_NAME.mk* or define $CLAS_CMS to point to your private copy.

If any of the above files had a header or include file located in, beneath or parallel directory *Makefile* would find it and add that directory to the include search path, defined by the variable `CLAS_INCLUDES`. You can add directories containing header or include files to `CLAS_INCLUDES` if they outside to scope of *Makefile*'s search path.

One reason *Makefile* is so large is that it is versatile and you are encouraged to use it with any of your projects. It is possible that you will not have to write a makefile again for C or FORTRAN compiling if you use this Makefile.

5.9 Machine Dependence

5.10 Custom Makefile requirements

5.11 Makefile Variables

In this section the user definable variables in *Makefile* are listed along with their function. The most often changed variables are listed first. If defaults are given, they can be overridden simply by defining the variable in your shell before invoking make.

TOP_DIR

default=/home/$USERNAME
defined in: *$CLAS_CMS/Makefile*
overridden if set in environment $TOP_DIR/bin/$OS_NAME and $TOP_DIR/lib/$OS_NAME are the locations for binaries and libraries. You can override the default by defining $TOP_DIR as an environment variable. For example: `setenv TOP_DIR /scratch/$USERNAME`.

LIBNAMES

default = none
defined in:
This is the link list that is fed to the `ld` linker. Default link lists can be found in *$CLAS_CMS/link.mk* and *$CLAS_CMS/link.$OS_NAME.mk*, where *$CLAS_CMS/link.$OS_NAME.mk* contains the machine dependent parts of the link list. See the example below on how these default link lists are used.

SHARED_LIBS

default= none
defined in:
This variable is needed due to a small bug in GNU Make (or maybe we missed a trick while reading the manual). The libraries defined in LIBNAMES are not only passed to the linker but also used for dependencies. Unfortunately when GNU Make receives `-lmycode` as a dependency it looks for *libmycode.a*. If *libmycod* is a shared library it will not have the **.a** extension but a **.so** extension, so shared libraries will fail the dependency test. To get around this the SHARED_LIBS variable was created. Libraries defined in SHARED_LIBS are **not** dependency tested, but simply fed to the linker.

LIB_PATH

default= $(locallib) $(CLAS_LIB) $(CERN_ROOT)/lib $(TCL_LIB) $(X_LIB)
defined in: *$CLAS_CMS/Makefile*
This is the directory search path for libraries during the link phase. If you wish to add a directory simply add the line, `LIB_PATH += newdirectory` and this directory will be added to the search path.

FFLAGS

default= machine dependent
defined in: *$CLAS_CMS/flags.$OS_NAME.mk*
FORTRAN compile flags, to add flags simply include the line `FFLAGS+= -mynewoptions` in the Makefile

CFLAGS

default= machine dependent
defined in: *$CLAS_CMS/flags.$OS_NAME.mk*
C compile flags, to add flags simply include the line `CFLAGS+= -mynewoptions` in the Makefile

CREATED_F
default = none
defined in: *cms/Makefile*
This variable is provided for those FORTRAN routines, usually **main** programs, that you need the object file on its own (not in the library). These object files are automatically added to the REQUIRED_OBJS list.

CREATED_C
default = none
defined in: *cms/Makefile*
This variable is provided for those C routines, usually **main** programs, that you need the object file on its own (not in the library). These object files are automatically added to the REQUIRED_OBJS list.

REQUIRED_OBJS
default=none
defined in: *cms/Makefile*
This variable is used to add *object* files to the link list. For example, the recsis link list requires two object files, recsis.o and rec_work.o and REQUIRED_OBJS is used to define these two object files.

CLAS_INCLUDES
default=$(LOCAL_INCLUDES) $(DEFAULT_INCLUDES)
defined in: *cms/Makefile*
This is the CPP include file search path. Where LOCAL_INCLUDES is defined dynamically via searches for include or header files and DEFAULT_INCLUDES=$CLAS_PACK/include $CLAS_PACK/gsim/include $TCL_INC $X_INC. You can add to or define your own CLAS_INCLUDES editing the line in the Makefile.

CLAS_CMS
default=none

defined in: environment
This environment variable is defined in *jlab.cshrc* and *yoursite.cshrc*. Redefine it if you are trying to customize or work on the makefile system. This variable points to the directory that contains the Makefiles.

CERN_ROOT/lib
default=none
defined in: environment
This environment variable should point to the location of the CERN object libraries.

TCL_LIB
default=/usr/local/lib (/usr/lib on Linux)
defined in: *$CLAS_CMS/flags.$OS_NAME.mk* **but** overridden if defined in the environment
This variable should point to the location of tcl object libraries.

TCL_INC
default=/usr/local/include (/usr/include on Linux)
defined in: *$CLAS_CMS/flags.$OS_NAME.mk* **but** overridden if defined in the environment
This variable should point to the tcl header files.

X_LIB
default=varies from machine to machine
defined in: *$CLAS_CMS/flags.$OS_NAME.mk* **but** overridden if defined in the environment
This variable should point to the X libraries.

ADD_DEBUG
default = "blank"
defined in:*$CLAS_CMS/flags.$OS_NAME.mk*
When the *Makefile* is invoked with the command line argument, DEBUG=yes, then ADD_DEBUG = _debug and `-g` is added to the FFLAGS and CFLAGS variable. The suffix, _debug, is added to the name of all CLAS libraries in the link list and to the created binary or created library name. Thus `make DEBUG=yes` in the *user* directory

results in a library named, *libuser_debug* and an executable *user_debug*.

5.12 The CLAS Environment Variables

As mentioned in the tutorials you need to `source yoursite.cshrc`. to define a set of environment variables used to compile, link and run CLAS software. The environment variables are defined here for completeness, those variable with "jlab definition" listings will need to be modified for offsite builds where as those with "definition" listings should be invariant.

OSCLAS set to the response of the `uname` command, used to define the bin and lib areas.

POSIX_SHELL /usr/bin/sh

CLAS_LOCATION defines the very top of the CLAS partition.
jlab definition = /apps/clas

CLAS_ROOT
jlab definition = $CLAS_LOCATION/u1

CVSROOT defines the location of the repository
jlab definition = $CLAS_ROOT/clas_cvs

BUILDS defines the top of builds directory underneath which you will find all the libraries, binaries and source.
definition = $CLAS_ROOT/builds

CLAS_PROD points to the PRODUCTION build
definition= $BUILDS/PRODUCTION

CLAS_DEVEL points to the development build
definition = $BUILDS/DEVELOPMENT

CLAS_LEVEL chooses the code level PRODUCTION or DEVELOPMENT
definition = user's choice

CLAS_LIB points to the library location
definition $BUILDS/$CLAS_LEVEL/lib/$OSCLAS

CLAS_BIN points to the binary location
definition = $BUILDS/$CLAS_LEVEL/bin/$OSCLAS

CLAS_PACK points to the top of the source tree
definition = $BUILDS/$CLAS_LEVEL/packages

CLAS_CMS points to the directory that contains the makefiles
definition = $CLAS_PACK/cms

CLAS_SCRIPTS points to the directory that holds utility scripts
definition = $CLAS_PACK/scripts

RECSIS same as CLAS_PACK needed until recsis code is changed to reflect the new directory structure.
definition = $CLAS_PACK

CLAS_DATA points to the location of data on disk
jlab definition = $CLAS_LOCATION/u2/data

CLAS_PARMS points to the calibration constants
jlab definition = $CLAS_ROOT/parms

RECSIS_RUNTIME points to the directory that holds the recsis sequence number file.
jlab definition = $CLAS_ROOT/clsrc/recsis/runtime

Another deviation from the previous system is that this file does not muck with your PATH. The PATH environment variable is something the user needs to have complete control over. We suggest that you place CLAS_BIN and CLAS_SCRIPTS into your path but leave it to the users to do this. It would also be convenient for the users if they would define TOP_DIR in their *.cshrc* to be the location of their libraries and binaries (if the default /home/$USERNAME is unacceptable). Once TOP_DIR is defined the user should probably place $TOP_DIR/bin/$OSCLAS into their PATH **before** $CLAS_BIN, that way they will pick up their executables without having to type in the full path. JLab users will also have to define the CERN environment variables themselves.

6 Code Management

6.1 Revision Control: CVS

We use CVS to manage the directory structure. Basically, an ascii file that can not be produced via a make step is kept under CVS version control. CVS has many features, and we refer you to the rather-well-written manual, but some features that are relevant here are:

Remote repository access. The repository can be accessed from off-site machines via the internet. This means that all collaborators can work from the same set of files.

RCS is used. CVS can be thought of as a fancy wrapper around RCS calls. All of the useful features of RCS are thus retained.

Version tagging. Once a coherent version of the tree has been obtained, it can be tagged such that subsequent checkouts can, if desired, reproduce the tagged version, even if in the interim modifications of the files have been checked into the repository.

Every CLAS collaborators that want to access the CLAS CVS repository needs to have an account on the JLab computers. This includes offsite workers. Even if offsite workers never actually run software on the JLab machines the JLab account is needed so they can check in or out code directly from the JLab repository. You should verify that they are a member of the "clas" group before proceeding. To verify this, enter `groups` and your response should include `clas`. If `clas` was not one of the groups listed you should contact the JLab Computer Center to be added to the `clas` group. If you are accessing the CVS repository remotely, you need to have clas as your default group at login. The first group in the response to the `groups` command is your default group.

6.2 The Repository and Builds

The CVS repository is where all version of all files for all time are stored. The head of the repository directory tree is $CVSROOT. The files in the tree are ascii and in RCS format. As such the files will not compile and are not easy to read. Only in rare cases will it become necessary to cd into the repository. You probably should not even consider it. And if you ever go in there, you can look but you better not touch!

What we call a complete build is a usable version of the tree, created initially by a CVS checkout, and fleshed out by the invocation of makefiles. Various versions of builds can be made from the same repository by changing the details of the CVS checkout command (or commands). At JLab we are planning to have at least two builds in existence at all time, named development and production.

6.2.1 Development Build

This will in principle be a reflection of the latest version of all files in the repository at the time of the build. Said another way, this build will contain everyone's latest changes. Scary, huh? The reality is that simply taking the latest and greatest may not produce a working tree. So in practice, some packages may have to be backed up in time (CVS can do this rather easily) until a working tree obtains. Developers of code which breaks the tree will be told to fix the problem and check in a functional version.

We expect to do a development build every week.

6.2.2 Production Build

Occasionally the tree will be in a state where things seem relatively bug free and we would like to declare it a stable version, fit for use by the casual user. This state may or may not contain the most-current versions; that is a detail. At these times, we can tag the entire tree and build a version based on the tag to create an official production build.

Production builds will be less volatile than development builds. Overall circumstances in the offline analysis will often dictate when a production build is needed. We can probably expect a new production build every month or so.

Note that production builds (and for that matter, development builds) are never completely "thrown away" even after they disappear from the disk. Any member of the collaboration, given sufficient disk space, should be able to replicate any given production build by using the appropriate tag when doing a CVS checkout from the repository.

6.2.3 Toggling between Production and Development

The *jlab.cshrc* file also defines two aliases `use_prod` and `use_dev` which will reset your environment to point to development, `use_dev` or production `use_prod`. You can invoke these aliases anytime you wish, if for example the development libraries break, type `use_prod` and switch to production.

6.3 Offsite University_of_higher_learning.cshrc

As mentioned in the Librarian Tutorial, one of the first things the off-site librarian should do after a successful build is to customize the *packages/cms/offsite.cshrc*. Customization will involve redefining site specific environment variables, such as $CLAS_ROOT and $CLAS_PARM Then they should rename it to reflect their institution, for example *uva.cshrc*, and then check it into the repository so that it will be there ready to go the next time a complete build is performed. The file *packages/cms/offsite.cshrc* is provided as a template.

6.4 Example: Complete Directory Tree at JLab

As an example of how to structure the totality of files needed to use the system, we show the structure we are currently using at JLab.

The top of the tree is */apps/clas/u1*.

```
/apps/clas/u1/
             |--clas_cvs/    # CVS repository. Do not go here.
             |--parms/       # Runtime parameters
             |--docs/        # Documentation
             |--builds/      # Top of the build branch
                      |--DEVELOPMENT/  # Softlink to apr.1.1997
                      |--PRODUCTION/  # Softlink to apr.26.1997
                      |--apr.1.1997/  # Current PRODUCTION
                      |            |--bin/
                      |            |--lib/
                      |            |--packages/
                      |
                      |--apr.26.1997/     # Current DEVELOPMENT
                      |--mar.28.1997/     # Obsolete build
```

6.5 CVS Commands

To give a flavor of the use of CVS, we list some of the commonly used CVS commands with an eye to this specific application.

checkout: Pulls files out of the repository. You can checkout out individual packages (e. g., `cvs checkout dc`) or the entire tree (`cvs checkout packages`).

commit: Put a modified version of a file into the repository, creating a new revision of that file.

status: Gives the status of a file under CVS control. Right now, if we look at the status of this document we see:

```
> cvs status cms.tex
File: cms.tex            Status: Locally Modified
```

Here "Locally Modified" means that the file has changed since its checkout. Other possible states are:

"Up-to-date": Just as checked out.

"Needs Checkout": The file has not been changed since checkout, but since then a newer revision has been committed to the repository.

"Needs Merge": The file has been modified since checkout and since then a newer revision has been committed to the repository. What then you ask? We are not going to get into that here. See the manual.

log: Gives the complete revision history of a file, including tags and the comments that accompanied each commit.

diff: Does a diff between the current version on disk and the version that was checked out. With switches on it can diff any combination of revisions and the current disk version.

release: Delete a module (a set of files), but before doing so check the status of all files in the module and prompt the user for confirmation before actually deleting them. If there are any files that have a status other than "Up-to-date" or if there are files found that are not under cvs control, the release command will note these before the prompt.

update: Bring a file up-to-date. This may involve bringing in a newer revision, merging a new revision with a modified local file (with or without conflicts), or nothing at all if the file is already up-to-date. This a bit tricky. See the manual.

add: Add a file to the repository.

remove: Remove a file from the repository. The file is not actually deleted. Rather a new revision is created with the attribute "dead". Subsequent checkouts will skip all dead file. The file is kept so that the full time history of the repository can be retained.

tag: Put a symbolic tag on a file or module.

6.6 Random Notes on Using the Repository

- Don't check in changes to user, unless it is a template.

- Only check in tested changes. The I-only-made-a-small-change-what-could-possibly-go-wrong attitude is a recipe for disaster.

- Use your own JLab account when you do remote `cvs` operations (unless you don't want to get caught). If you use the account `clasrun`, the history file shows `clasrun` as the modifier of the repository, and we can't trace changes back to the perpetrator.

- To change your default group, use the `newgrp` command. This puts you in a new shell with the requested default group. This is not a permanent fix. The next time you log in from scratch, you'll have to do it again. Contact the computer center to make `clas` your "permanent" default group. You can fix files written with the wrong group id with the `chgrp` command.

- Make sure your `umask` is 2 before writing public files, so others can modify the files. Do a `umask` to see what yours is and do a `man umask` to find out what it means.

6.7 The Need for a CLAS librarian

In order for the CLAS software to be maintained a code librarian must exist. The job would entail making weekly builds, verifying code integrity, bouncing code back to developers if their code does not work, communicating with offsite builders.... This job should take about half a day of effort per week. As a librarian you will learn the intimate details of the CLAS code while contributing an immense service to the collaboration. Typically the job should rotate on a yearly basis. If an collaborating institution wants to ensure that they have a "clas code" expert within their ranks it is to their advantage to provide a graduate student or post-doc to perform this function. **Without a librarian this system as well as any other system fails.**

References

[1] GNU Make: A Program for Directing Recompilation. Richard M. Stallman and Roland McGrath. Free Software Foundation. 1996.

[2] Version Management with CVS. Per Cederqvist et al. Signum Support AB. 1993.